# 字节对齐方式

一、快速理解

1. 什么是字节对齐？

在C语言中，结构是一种复合数据类型，其构成元素既可以是基本数据类型（如int、long、float等）的变量，也可以是一些复合数据类型（如数组、结构、联合等）的数据单元。在结构中，编译器为结构的每个成员按其自然边界（alignment）分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同。

为了使CPU能够对变量进行快速的访问,变量的起始地址应该具有某些特性,即所谓的”对齐”. 比如4字节的int型,其起始地址应该位于4字节的边界上,即起始地址能够被4整除.

2. 字节对齐有什么作用？

字节对齐的作用不仅是便于cpu快速访问，同时合理的利用字节对齐可以有效地节省存储空间。

对于32位机来说，4字节对齐能够使cpu访问速度提高，比如说一个long类型的变量，如果跨越了4字节边界存储，那么cpu要读取两次，这样效率就低了。但是在32位机中使用1字节或者2字节对齐，反而会使变量访问速度降低。所以这要考虑处理器类型，另外还得考虑编译器的类型。在vc中默认是4字节对齐的，GNU gcc 也是默认4字节对齐。

3. 更改C编译器的缺省字节对齐方式

在缺省情况下，C编译器为每一个变量或是数据单元按其自然对界条件分配空间。一般地，可以通过下面的方法来改变缺省的对界条件：

· 使用伪指令#pragma pack (n)，C编译器将按照n个字节对齐。

· 使用伪指令#pragma pack ()，取消自定义字节对齐方式。

另外，还有如下的一种方式：

· \_\_attribute((aligned (n)))，让所作用的结构成员对齐在n字节自然边界上。如果结构中有成员的长度大于n，则按照最大成员的长度来对齐。

· \_\_attribute\_\_ ((packed))，取消结构在编译过程中的优化对齐，按照实际占用字节数进行对齐。

4. 举例说明

例1

struct test

{

char x1;

short x2;

float x3;

char x4;

};

由于编译器默认情况下会对这个struct作自然边界（有人说“自然对界”我觉得边界更顺口）对齐，结构的第一个成员x1，其偏移地址为0，占据了第1个字节。第二个成员x2为short类型，其起始地址必须2字节对界，因此，编译器在x2和x1之间填充了一个空字节。结构的第三个成员x3和第四个成员x4恰好落在其自然边界地址上，在它们前面不需要额外的填充字节。在test结构中，成员x3要求4字节对界，是该结构所有成员中要求的最大边界单元，因而test结构的自然对界条件为4字节，编译器在成员x4后面填充了3个空字节。整个结构所占据空间为12字节。

例2

#pragma pack(1) //让编译器对这个结构作1字节对齐

struct test

{

char x1;

short x2;

float x3;

char x4;

};

#pragma pack() //取消1字节对齐，恢复为默认4字节对齐

这时候sizeof(struct test)的值为8。

例3

#define GNUC\_PACKED \_\_attribute\_\_((packed))

struct PACKED test

{

char x1;

short x2;

float x3;

char x4;

}GNUC\_PACKED;

这时候sizeof(struct test)的值仍为8。

二、深入理解

什么是字节对齐,为什么要对齐?

TragicJun 发表于 2006-9-18 9:41:00现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的CPU在访问一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。显然在读取效率上下降很多。

二.字节对齐对程序的影响:

先让我们看几个例子吧(32bit,x86环境,gcc编译器):

设结构体如下定义：

struct A

{

int a;

char b;

short c;

};

struct B

{

char b;

int a;

short c;

};

现在已知32位机器上各种数据类型的长度如下:

char:1(有符号无符号同)

short:2(有符号无符号同)

int:4(有符号无符号同)

long:4(有符号无符号同)

float:4 double:8

那么上面两个结构大小如何呢?

结果是:

sizeof(strcut A)值为8

sizeof(struct B)的值却是12

结构体A中包含了4字节长度的int一个，1字节长度的char一个和2字节长度的short型数据一个,B也一样;按理说A,B大小应该都是7字节。

之所以出现上面的结果是因为编译器要对数据成员在空间上进行对齐。上面是按照编译器的默认设置进行对齐的结果,那么我们是不是可以改变编译器的这种默认对齐设置呢,当然可以.例如:

#pragma pack (2) /\*指定按2字节对齐\*/

struct C

{

char b;

int a;

short c;

};

#pragma pack () /\*取消指定对齐，恢复缺省对齐\*/

sizeof(struct C)值是8。

修改对齐值为1：

#pragma pack (1) /\*指定按1字节对齐\*/

struct D

{

char b;

int a;

short c;

};

#pragma pack () /\*取消指定对齐，恢复缺省对齐\*/

sizeof(struct D)值为7。

后面我们再讲解#pragma pack()的作用.

三.编译器是按照什么样的原则进行对齐的?

先让我们看四个重要的基本概念：

1.数据类型自身的对齐值：

对于char型数据，其自身对齐值为1，对于short型为2，对于int,float,double类型，其自身对齐值为4，单位字节。

2.结构体或者类的自身对齐值：其成员中自身对齐值最大的那个值。

3.指定对齐值：#pragma pack (value)时的指定对齐值value。

4.数据成员、结构体和类的有效对齐值：自身对齐值和指定对齐值中小的那个值。

有了这些值，我们就可以很方便的来讨论具体数据结构的成员和其自身的对齐方式。有效对齐值N是最终用来决定数据存放地址方式的值，最重要。有效对齐N，就是表示“对齐在N上”，也就是说该数据的"存放起始地址%N=0".而数据结构中的数据变量都是按定义的先后顺序来排放的。第一个数据变量的起始地址就是数据结构的起始地址。结构体的成员变量要对齐排放，结构体本身也要根据自身的有效对齐值圆整(就是结构体成员变量占用总长度需要是对结构体有效对齐值的整数倍，结合下面例子理解)。这样就不能理解上面的几个例子的值了。

例子分析：

分析例子B；

struct B

{

char b;

int a;

short c;

};

假设B从地址空间0x0000开始排放。该例子中没有定义指定对齐值，在笔者环境下，该值默认为4。第一个成员变量b的自身对齐值是1，比指定或者默认指定对齐值4小，所以其有效对齐值为1，所以其存放地址0x0000符合0x0000%1=0.第二个成员变量a，其自身对齐值为4，所以有效对齐值也为4，所以只能存放在起始地址为0x0004到0x0007这四个连续的字节空间中，复核0x0004%4=0,且紧靠第一个变量。第三个变量c,自身对齐值为2，所以有效对齐值也是2，可以存放在0x0008到0x0009这两个字节空间中，符合0x0008%2=0。所以从0x0000到0x0009存放的都是B内容。再看数据结构B的自身对齐值为其变量中最大对齐值(这里是b）所以就是4，所以结构体的有效对齐值也是4。根据结构体圆整的要求，0x0009到0x0000=10字节，（10＋2）％4＝0。所以0x0000A到0x000B也为结构体B所占用。故B从0x0000到0x000B共有12个字节,sizeof(struct B)=12;其实如果就这一个就来说它已将满足字节对齐了,因为它的起始地址是0,因此肯定是对齐的,之所以在后面补充2个字节,是因为编译器为了实现结构数组的存取效率,试想如果我们定义了一个结构B的数组,那么第一个结构起始地址是0没有问题,但是第二个结构呢?按照数组的定义,数组中所有元素都是紧挨着的,如果我们不把结构的大小补充为4的整数倍,那么下一个结构的起始地址将是0x0000A,这显然不能满足结构的地址对齐了,因此我们要把结构补充成有效对齐大小的整数倍.其实诸如:对于char型数据，其自身对齐值为1，对于short型为2，对于int,float,double类型，其自身对齐值为4，这些已有类型的自身对齐值也是基于数组考虑的,只是因为这些类型的长度已知了,所以他们的自身对齐值也就已知了.

同理,分析上面例子C：

#pragma pack (2) /\*指定按2字节对齐\*/

struct C

{

char b;

int a;

short c;

};

#pragma pack () /\*取消指定对齐，恢复缺省对齐\*/

第一个变量b的自身对齐值为1，指定对齐值为2，所以，其有效对齐值为1，假设C从0x0000开始，那么b存放在0x0000，符合0x0000%1=0;第二个变量，自身对齐值为4，指定对齐值为2，所以有效对齐值为2，所以顺序存放在0x0002、0x0003、0x0004、0x0005四个连续字节中，符合0x0002%2=0。第三个变量c的自身对齐值为2，所以有效对齐值为2，顺序存放

在0x0006、0x0007中，符合0x0006%2=0。所以从0x0000到0x00007共八字节存放的是C的变量。又C的自身对齐值为4，所以C的有效对齐值为2。又8%2=0,C只占用0x0000到0x0007的八个字节。所以sizeof(struct C)=8.

四.如何修改编译器的默认对齐值?

1.在VC IDE中，可以这样修改：[Project]|[Settings],c/c++选项卡Category的Code Generation选项的Struct Member Alignment中修改，默认是8字节。

2.在编码时，可以这样动态修改：#pragma pack .注意:是pragma而不是progma.

五.针对字节对齐,我们在编程中如何考虑?

如果在编程的时候要考虑节约空间的话,那么我们只需要假定结构的首地址是0,然后各个变量按照上面的原则进行排列即可,基本的原则就是把结构中的变量按照类型大小从小到大声明,尽量减少中间的填补空间.还有一种就是为了以空间换取时间的效率,我们显示的进行填补空间进行对齐,比如:有一种使用空间换时间做法是显式的插入reserved成员：

struct A{

char a;

char reserved[3];//使用空间换时间

int b;

}

reserved成员对我们的程序没有什么意义,它只是起到填补空间以达到字节对齐的目的,当然即使不加这个成员通常编译器也会给我们自动填补对齐,我们自己加上它只是起到显式的提醒作用.

六.字节对齐可能带来的隐患:

代码中关于对齐的隐患，很多是隐式的。比如在强制类型转换的时候。例如：

unsigned int i = 0x12345678;

unsigned char \*p=NULL;

unsigned short \*p1=NULL;

p=&i;

\*p=0x00;

p1=(unsigned short \*)(p+1);

\*p1=0x0000;

最后两句代码，从奇数边界去访问unsignedshort型变量，显然不符合对齐的规定。

在x86上，类似的操作只会影响效率，但是在MIPS或者sparc上，可能就是一个error,因为它们要求必须字节对齐.

七.如何查找与字节对齐方面的问题:

如果出现对齐或者赋值问题首先查看

1. 编译器的big little端设置

2. 看这种体系本身是否支持非对齐访问

3. 如果支持看设置了对齐与否,如果没有则看访问时需要加某些特殊的修饰来标志其特殊访问操作。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# C/C++内存对齐

一、什么是字节对齐,为什么要对齐?

现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特 定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的CPU在访问 一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对 数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那 么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数 据。显然在读取效率上下降很多。

二、请看下面的结构：

struct MyStruct

{

double dda1;

char dda;

int type

};

对结构MyStruct采用sizeof会出现什么结果呢？sizeof(MyStruct)为多少呢？也许你会这样求：

sizeof(MyStruct)=sizeof(double)+sizeof(char)+sizeof(int)=13

但是当在VC中测试上面结构的大小时，你会发现sizeof(MyStruct)为16。你知道为什么在VC中会得出这样一个结果吗？

其实，这是VC对变量存储的一个特殊处理。为了提高CPU的存储速度，VC对一些变量的起始地址做了“对齐”处理。在默认情况下，VC规定各成员变量存放的起始地址相对于结构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数。下面列出常用类型的对齐方式(vc6.0,32位系统)。

类型

对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量）

Char

偏移量必须为sizeof(char)即1的倍数

int

偏移量必须为sizeof(int)即4的倍数

float

偏移量必须为sizeof(float)即4的倍数

double

偏移量必须为sizeof(double)即8的倍数

Short

偏移量必须为sizeof(short)即2的倍数

各成员变量在存放的时候根据在结构中出现的顺序依次申请空间，同时按照上面的对齐方式调整位置，空缺的字节VC会自动填充。同时VC为了确保结构的大小为结构的字节边界数（即该结构中占用最大空间的类型所占用的字节数）的倍数，所以在为最后一个成员变量申请空间后，还会根据需要自动填充空缺的字节。

下面用前面的例子来说明VC到底怎么样来存放结构的。

struct MyStruct

{

double dda1;

char dda;

int type

}；

为上面的结构分配空间的时候，VC根据成员变量出现的顺序和对齐方式，先为第一个成员dda1分配空间，其起始地址跟结构的起始地址相同（刚好偏移量0刚好为sizeof(double)的倍数），该成员变量占用sizeof(double)=8个字节；接下来为第二个成员dda分配空间，这时下一个可以分配的地址对于结构的起始地址的偏移量为8，是sizeof(char)的倍数，所以把dda存放在偏移量为8的地方满足对齐方式，该成员变量占用 sizeof(char)=1个字节；接下来为第三个成员type分配空间，这时下一个可以分配的地址对于结构的起始地址的偏移量为9，不是sizeof (int)=4的倍数，为了满足对齐方式对偏移量的约束问题，VC自动填充3个字节（这三个字节没有放什么东西），这时下一个可以分配的地址对于结构的起始地址的偏移量为12，刚好是sizeof(int)=4的倍数，所以把type存放在偏移量为12的地方，该成员变量占用sizeof(int)=4个字节；这时整个结构的成员变量已经都分配了空间，总的占用的空间大小为：8+1+3+4=16，刚好为结构的字节边界数（即结构中占用最大空间的类型所占用的字节数sizeof(double)=8）的倍数，所以没有空缺的字节需要填充。所以整个结构的大小为：sizeof(MyStruct)=8+1+ 3+4=16，其中有3个字节是VC自动填充的，没有放任何有意义的东西。

下面再举个例子，交换一下上面的MyStruct的成员变量的位置，使它变成下面的情况：

struct MyStruct

{

char dda;

double dda1;

int type

}；

这个结构占用的空间为多大呢？在VC6.0环境下，可以得到sizeof(MyStruc)为24。结合上面提到的分配空间的一些原则，分析下VC怎么样为上面的结构分配空间的。（简单说明）

struct MyStruct

{

char dda; //偏移量为0，满足对齐方式，dda占用1个字节；

double dda1;//下一个可用的地址的偏移量为1，不是sizeof(double)=8

//的倍数，需要补足7个字节才能使偏移量变为8（满足对齐

//方式），因此VC自动填充7个字节，dda1存放在偏移量为8

//的地址上，它占用8个字节。

int type； //下一个可用的地址的偏移量为16，是sizeof(int)=4的倍

//数，满足int的对齐方式，所以不需要VC自动填充，type存

//放在偏移量为16的地址上，它占用4个字节。

}；//所有成员变量都分配了空间，空间总的大小为1+7+8+4=20，不是结构

//的节边界数（即结构中占用最大空间的类型所占用的字节数sizeof

//(double)=8）的倍数，所以需要填充4个字节，以满足结构的大小为

//sizeof(double)=8的倍数。

所以该结构总的大小为：sizeof(MyStruc)为1+7+8+4+4=24。其中总的有7+4=11个字节是VC自动填充的，没有放任何有意义的东西。

VC对结构的存储的特殊处理确实提高CPU存储变量的速度，但是有时候也带来了一些麻烦，我们也屏蔽掉变量默认的对齐方式，自己可以设定变量的对齐方式。

VC 中提供了#pragma pack(n)来设定变量以n字节对齐方式。n字节对齐就是说变量存放的起始地址的偏移量有两种情况：第一、如果n大于等于该变量所占用的字节数，那么偏移量必须满足默认的对齐方式，第二、如果n小于该变量的类型所占用的字节数，那么偏移量为n的倍数，不用满足默认的对齐方式。结构的总大小也有个约束条件，分下面两种情况：如果n大于所有成员变量类型所占用的字节数，那么结构的总大小必须为占用空间最大的变量占用的空间数的倍数；

否则必须为n的倍数。下面举例说明其用法。

#pragma pack(push) //保存对齐状态

#pragma pack(4)//设定为4字节对齐

struct test

{

char m1;

double m4;

int m3;

};

#pragma pack(pop)//恢复对齐状态

以上结构的大小为16，下面分析其存储情况，首先为m1分配空间，其偏移量为0，满足我们自己设定的对齐方式（4字节对齐），m1占用1个字节。接着开始为 m4分配空间，这时其偏移量为1，需要补足3个字节，这样使偏移量满足为n=4的倍数（因为sizeof(double)大于n）,m4占用8个字节。接着为m3分配空间，这时其偏移量为12，满足为4的倍数，m3占用4个字节。这时已经为所有成员变量分配了空间，共分配了16个字节，满足为n的倍数。如果把上面的#pragma pack(4)改为#pragma pack(16)，那么我们可以得到结构的大小为24。（请读者自己分析）

三、再看下面这个例子

#pragma pack(8)

struct S1{

char a;

long b;

};

struct S2 {

char c;

struct S1 d;

long long e;

};

#pragma pack()

sizeof(S2)结果为24.

成员对齐有一个重要的条件,即每个成员分别对齐.即每个成员按自己的方式对齐.

也就是说上面虽然指定了按8字节对齐,但并不是所有的成员都是以8字节对齐.其对齐的规则是,每个成员按其类型的对齐参数(通常是这个类型的大小)和指定对齐参数(这里是8字节)中较小的一个对齐.并且结构的长度必须为所用过的所有对齐参数的整数倍,不够就补空字节.

S1中,成员a是1字节默认按1字节对齐,指定对齐参数为8,这两个值中取1,a按1字节对齐;成员b是4个字节,默认是按4字节对齐,这时就按4字节对齐,所以sizeof(S1)应该为8;

S2 中,c和S1中的a一样,按1字节对齐,而d 是个结构,它是8个字节,它按什么对齐呢?对于结构来说,它的默认对齐方式就是它的所有成员使用的对齐参数中最大的一个,S1的就是4.所以,成员d就是按4字节对齐.成员e是8个字节,它是默认按8字节对齐,和指定的一样,所以它对到8字节的边界上,这时,已经使用了12个字节了,所以又添加了4个字节的空,从第16个字节开始放置成员e.这时,长度为24,已经可以被8(成员e按8字节对齐)整除.这样,一共使用了24个字节.

a b

S1的内存布局：11\*\*,1111,

c S1.a S1.b d

S2的内存布局：1\*\*\*,11\*\*,1111,\*\*\*\*11111111

这里有三点很重要:

1.每个成员分别按自己的方式对齐,并能最小化长度。

2.复杂类型(如结构)的默认对齐方式是它最长的成员的对齐方式,这样在成员是复杂类型时,可以最小化长度。

3.对齐后的长度必须是成员中最大的对齐参数的整数倍,这样在处理数组时可以保证每一项都边界对齐。

Win32平台下的微软 编译器(cl.exe for 80×86)的对齐策略：

1) 结构体变量的首地址能够被其最宽基本类型成员的大小所整除；

备注：编译器在给结构体开辟空间时，首先找到结构体中最宽的基本数据类型，然后寻找内存地址能被该基本数据类型所整除的位置，作为结构体的首地址。将这个最宽的基本数据类型的大小作为上面介绍的对齐模数。

2) 结构体每个成员相对于结构体首地址的偏移量（offset）都是成员大小的整数倍，如有需要编译器会在成员之间加上填充字节（internal adding）；

备注:为结构体的一个成员开辟空间之前，编译器首先检查预开辟空间的首地址相对于结构体首地址的偏移是否是本成员的整数倍，若是，则存放本成员，反之，则在本成员和上一个成员之间填充一定的字节，以达到整数倍的要求，也就是将预开辟空间的首地址后移几个字节。

3) 结构体的总大小为结构体最宽基本类型成员大小的整数倍，如有需要，编译器会在最末一个成员之后加上填充字节（trailing padding）。

备注：结构体总大小是包括填充字节，最后一个成员满足上面两条以外，还必须满足第三条，否则就必须在最后填充几个字节以达到本条要求。

